

Software Methodologies in VHDL Code Analysis

Cristiana Bolchini and Luciano Baresi
Dipartimento di Elettronica e Informazione
Politecnico di Milano
P.zza L. Da Vinci, 32 - 20133 Milano, Italy
[bolchini|baresi]@elet.polimi.it

Abstract

At a high level of abstraction, the VHDL specification of the functionalities that a circuit shall perform is given by defining the behavioral model. The similarity with procedural programming languages suggested to tailor some software analysis techniques to VHDL behavioral description analysis. The aim is to retrieve information on the final circuit from its specifications. The paper presents several analyses of the code aimed at identifying significant properties of the final circuit from the synthesis and testability points of view.

1 Introduction

VHDL is now the most widespread language for building software models of hardware systems. However, although the complexity of the described models is rapidly increasing, no support is given to designers to evaluate the quality of their code and of results achieved by its synthesis. A similar problem has been already investigated by software engineers leading to the definition of several techniques for analyzing and evaluating software specifications [1, 2, 3].

The aim of this paper is to consider static analysis techniques to identify significant properties of the implementation, from the behavioral VHDL description. The information collected from the code analysis is very important for:

- discovering and correcting possible errors, inconsistencies, and incompletenesses, without affecting the following design phases. In this way, backtracks are minimized, saving resources and avoiding last-minute solutions, that could jeopardize the efficiency and original functionalities of the model.

- deriving test patterns, that will be used to validate the circuit during gate (or switch) level testing.
- identifying code fragments, that are known to be critical with respect to synthesis. The adoption of different guidelines in writing a specification leads to different circuit realizations. Thus, users can heavily guide subsequent synthesis by adopting different styles in writing their code.
- predicting the outputs of a synthesis tool, i.e., how a synthesized circuit should look like. Theoretical results are then compared with actual results, allowing users to get acquainted with the instrument to better “control” the obtained results.

Approaches to VHDL analysis for quality evaluation have been already published in literature [4, 5, 6, 7]. New commercial tools mainly dealing with code coverage are already available [8, 9]. More sophisticated analyses for synthesizability, simulation, complexity evaluation and testability are under study in the EEC supported OMI Project REQUEST.

This paper presents the first results of this project by proposing a data flow based methodology to statically extract information from the VHDL specification, which is interesting both for testability analysis and for synthesis purposes.

The syntactic similarity between behavioral VHDL and procedural programming languages suggested to try to apply software analysis techniques to investigate VHDL code. The work describes an approach for identifying deadlock conditions within VHDL specifications. Hints and ideas have been taken from reachability analysis [3] and symbolic execution [10]. While proposing this technique, we adapted the concept of deadlock to the hardware domain; it corresponds to a situation where the system does not stop working while waiting for an event to occur, rather remains in the same state forever, producing the same values on the observable outputs. If the deadlock derives from an erroneous specification, the analysis allows the modification of the VHDL code, if it is a correct functional situation, the analysis may suggest the introduction of detection logic or specific reset logic. In both cases it is significant for the user to identify the possible deadlock condition, either for eliminating it or for ensuring the possibility to exit from such a condition. Besides this, several techniques based on data flow analysis are also presented to highlight useful properties of VHDL behavioral descriptions, focusing on the sequential aspects of the language.

The rest of this paper is organized as follows. Section 2 addresses deadlocks in hardware systems. By starting from the classical dining philosophers problem, we

tried to generalize the approach with respect to the problems concerning information exchanged among processes. Static analysis techniques, all based on some annotations of the flow graph of the VHDL code under analysis, are described in Section 3. Finally, Section 4 draws some conclusions, also providing further hints for analogies between analysis techniques for software specification and VHDL behavioral description analysis.

2 Deadlocks

Quoting [11], deadlocks can be defined as follows:

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Because all processes are waiting, none of them will ever cause any event, and all the processes remain blocked. This definition applies directly to hardware systems modeled by means of interacting VHDL processes. Though, differently from software systems:

- An event is a change on a signal. A process can either be suspended waiting for an event or need an event to change its outputs. In both cases, events are necessary to allow a circuit to evolve.
- A blocked circuit does not stop working. It evolves while remaining in the same state, producing always the same results. This condition is easily detectable by an external observer who always sees the same output values.
- Communications in VHDL are asynchronous, based on events on signals. As to model execution, during the start-up phase, all processes are executed either till the end or till they are suspended waiting for some events to occur. After that, processes are executed each time there is a change on the signals they are sensible to.
- There is no run-time support that defines the actual scheduling among processes. The dynamics of a system either is hard-coded within the specification (closed systems) or depends on the arrival of external signals (open systems).

The last point implies that the *possibility* for a closed system to enter a deadlock condition is a *certainty*. Thus a VHDL behavioral description of a closed system with deadlock should be always synthesized by a set of constant values, instead of a real

logic circuit. In the case of open systems, a deadlock derives from the timing of external inputs. The absence of a reset signal may become a critical point. In fact, the reset signal would provide the only means for exiting the deadlock situation, placing the circuit in its initial state.

Now, the problem is how deadlocks can be uncovered in hardware circuits and, even better, in VHDL behavioral specifications. The condition quoted so far is easily detectable by simulation, but, to the best of authors' knowledge, not by the analysis of designed models. Moreover, when considering systems characterized by a deadlock, even if the simulation highlighted constant outputs in the time domain (after a first initialization phase), the synthesis, using commercial logic synthesis tools [8, 12] did not produce simple constants (including set logic), but an unexpected logical circuit. This is due to the fact that having defined different processes, different finite state machines are synthesized. These finite state machines, possibly, share the logic but not the states due to the implied complexity of obtaining the resulting product machine during synthesis. Therefore, deadlock analysis can be effective to evaluate the quality of the design with respect to the specific problem.

The proposed analysis formalizes the mutual dependencies among events. It is not enough to study dependencies among processes, since each process can wait for events on different signals. *Dependencies graphs* code this information. Nodes are the signals exchanged among processes and edges define dependencies among them. An edge between a node s and a node t means that there is a change on s only if there is a change on t . A necessary condition to discover deadlocks is, thus, the presence of loops within the graph.

Dependencies graphs apply to both closed systems and open systems. When an open system is taken into account, its dependencies graph comprises dangling edges entering those nodes (signals) sensible to external events. In this way we model processes sensitivity to unpredictable signals. These new edges, however, do not impact on the dependencies, circularities, between signals.

We categorize the circumstances leading to deadlocks into two groups:

- problems due to synchronization among processes. In this case, involved processes exchange only boolean-like values to guide their behaviors. A rendezvous-like communication protocol is explicitly programmed.
- Problems due to the values that flow among processes. A more general class of problems addresses all the values exchanged, without constraining the analysis to boolean-like values only.

The former set of models is exemplified in Section 2.1, using the *dining philosophers problem*, a well-known problem as far as software systems are concerned. The latter class is studied in Section 2.2, by improving dependences graphs and exploiting ideas from symbolic execution [10].

2.1 A VHDL Model of the Dining Philosophers

Dijkstra's *Dining Philosophers problem* [13], although not a very realistic problem, does contain a non trivial deadlock and is probably the most commonly analyzed example in studying synchronization among Ada tasks [14, 15, 16].

The problem can be formulated as follows:

Five philosophers are seated around a table. Between each philosopher there is a single fork. The life of a philosopher consists of periods of eating and thinking. When a philosopher gets hungry, he tries to get his left and right fork, one at a time, in either order. If successful in acquiring the two forks, he eats for a while, then puts down the forks and thinks.

A problem can arise if each philosopher simultaneously grabs his left (right) fork and then waits for his right (left) fork. Since right (left) forks are not available, all philosophers starve and a deadlock occurs. Even if philosophers get their forks at different times, in the worst case, the situation described so far can happen again.

By looking at the code, presented in Figure 1, we notice that:

- two distinct signals (FORKNL and FORKMR) are used to represent the request for acquiring a fork: one for each adjacent philosopher. This solution has been adopted not to have multiple-driven signals, that leads to not synthesizable code.
- The actual availability of a fork is represented by a boolean signal (FORKN).
- All the philosophers are controlled by the same external clock, i.e., they all obey the same timing. The adoption of more than one clock would have led to not synthesizable models for commercial logic synthesis tools [8, 12].
- It is supposed that all the philosophers try to start eating at the same time. This is the easiest choice to have a deadlock. Moreover it does not require the use of external signals to control the philosophers, i.e., we have a closed system.

Figure 1 is only an excerpt of the whole model we defined. Instead of showing all the processes, we concentrate only on a “philosopher” and a “fork”. Both philosophers and forks differ among them only for the indexes that identify used signals and variables.

```

package DINING_PHILOSOPHERS_PACKAGE is
    type PHIL_STATUS is (STARVE, EAT);
end DINING_PHILOSOPHERS_PACKAGE;

entity DINING_PHILOSOPHERS is
    port(clk: in std_ulogic;
         P1, P2, P3, P4, P5: out PHIL_STATUS);
end DINING_PHILOSOPHERS;

architecture ARCH of DINING_PHILOSOPHERS is

    type STATE is (S0L, S0R, S1, S2, S3, S4, S5);
    signal FORK1, FORK2, FORK3, FORK4, FORK5: boolean := true;
    signal FORK1L, FORK1R, FORK2L, FORK2R, FORK3L: boolean := true;
    signal FORK3R, FORK4L, FORK4R, FORK5L, FORK5R: boolean := true;
    signal STATUS1, STATUS2, STATUS3, STATUS4, STATUS5: state;

begin
    PHILOSOPHER1: process
    begin
        wait until clk = '1' and clk'event;
        case STATUS1 is
            when S0L => if (FORK5 = true) then
                FORK1L <= false;
                STATUS1 <= S0R;
                P1 <= STARVE;
            end if;
            when S0R => if (FORK1 = true) then
                FORK1R <= false;
                STATUS1 <= S1;
                P1 <= EAT;
            end if;
            when S1 => STATUS1 <= S2;
            when S2 => STATUS1 <= S3;
            when S3 => STATUS1 <= S4;
            when S4 => STATUS1 <= S5;
            when S5 => P1 <= STARVE;
                STATUS1 <= S0L;
                FORK1L <= true;
                FORK1R <= true;
        end case
    end process PHILOSOPHER1;

    FORK1: process(FORK1R, FORK2L)
    begin
        if ((FORK1R = false) OR (FORK2L = false)) then
            FORK1 <= false;
        else
            FORK1 <= true;
        end if;
    end process FORK1;
    ...
end ARCH;

```

Figure 1: Dining Philosophers: a VHDL model

The dependencies graph for the philosophers problem is shown in Figure 2. For the sake of simplicity, we consider two philosophers only. It can be noticed that an event on FORK1R requires an event on FORK1, and an event on FORK1 requires an event on either FORK1R or FORK2L. The same circularities exist also among signals FORK1L, FORK2R, and FORK2. This means that the whole system could enter a circular wait condition, i.e., a deadlock condition.

More precisely, before the dummy execution (which performs the initialization of the variables and signals, determining the values that will remain constant in case of a deadlock) the condition is necessary. After the first execution, the condition identifying deadlocks is not necessary anymore, but it is turned into a sufficient condition.

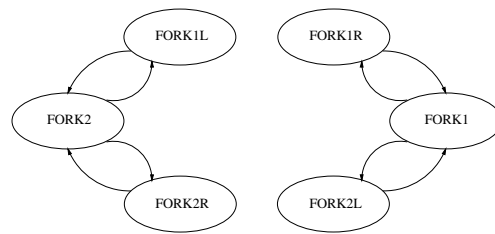


Figure 2: Philosophers: dependencies graph

The use of boolean signals to regulate both the request and the acquisition of a fork imposes a rendezvous-like communication model. This means that we can study how the model evolves in its execution space, borrowing some ideas from reachability analysis [3].

By looking at how signals are used we can define four high-level functions:

setRequest: a process P probes the variable v and sends a request to set v .

resetRequest: a process P sends a request to reset v .

set: a process P sets the variable v .

reset: a process P resets the variable v .

A **setRequest** on variable v fails if v is already set. In the same way, a **resetRequest** on v fails if v is not set.

The identification of these macro-instructions allows us to extract a graph representation of the processes, shown in Figure 3 (the numbers by the graph represent the labels of the states used in the execution space), that resembles *flowgraphs* [17]. According to the VHDL model, Figure 1, the graph for a philosopher process translates the finite state machine coded by the case statement. In the same way, the graph of a

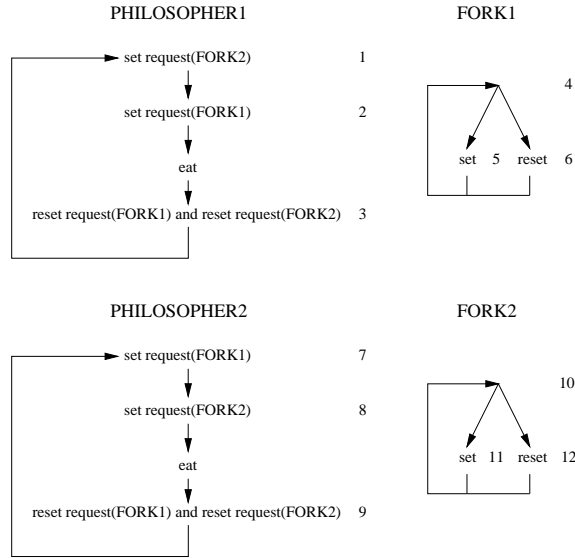


Figure 3: Philosophers: graph representations.

fork represents the two alternatives offered by an `if` statement. In both cases the loop is required by the dynamic semantics of VHDL.

These four graphs help us in building the execution space, Figure 4, of the model. VHDL processes execute and, if possible, change their state all in parallel. As expected, it is not possible to leave the state where each philosopher tries to get the second fork. The dashed line indicates that the whole system is not blocked, but it goes on executing in the same state.



Figure 4: Philosophers: execution space

2.2 Another Example

A simple, but significant case of deadlock due to the particular information exchanged among processes is exemplified by the fragment of VHDL code of Figure 5A. It is a divider taken from [18]. Consider the original model and a slightly changed version: $M \leq A \bmod B$ becomes $M \leq A + B$. Their dependencies graph is shown in Figure 5B.

It is possible to detect that, even if the dependencies graph highlights a circularity, only the original model enters a deadlock condition. In the first case, $A \bmod B$, where B is a given constant, returns the result M the first time. Then M is assigned to A

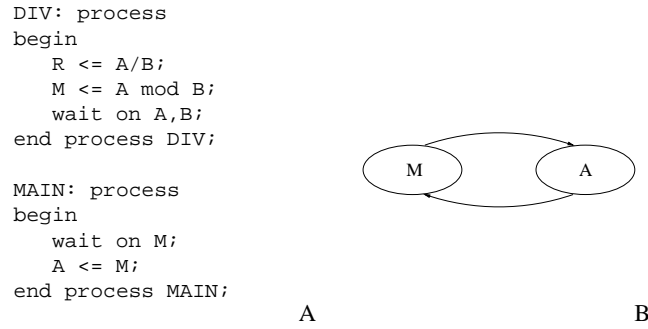


Figure 5: Divider: VHDL code

and, being $A = M$ less than B , the result of $A \text{ mod } B$ does not change anymore. M becomes constant and process MAIN remains waiting on M . The modified process, on the contrary, adds B to A , the old value of M , at each execution. Thus, M always increases, and the system never deadlocks.

The dependencies graph has to be augmented to provide a sufficient condition to detect this kind of deadlocks. Consider the modified dependencies graphs of Figure 6. Each node (signal) is associated with the function defining its value. Besides specifying the dependencies among signals, we code also the kind of dependency, that is the function “linking” the signals.

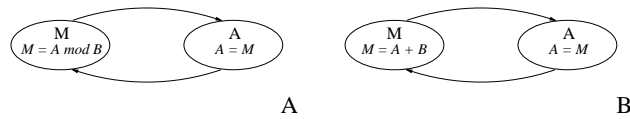


Figure 6: Modified dependencies graphs

Given this representation, we can define the systems of equations/inequations associated with the loop¹:

¹Notice that X' stands for the old value of X .

$$\left\{ \begin{array}{l} M = A \bmod B \\ M \neq M' \\ M' = A' \bmod B' \\ A = M \\ A \neq A' \\ A' = M' \end{array} \right. \quad \left\{ \begin{array}{l} M = A + B \\ M \neq M' \\ M' = A' + B' \\ A = M \\ A \neq A' \\ A' = M' \end{array} \right.$$

The solution of these systems would give the values of the exchanged signals for which the model does not enter a deadlock condition. The system can be deeply simplified by studying the opposite problem: the values of the involved variables/signals for which the loop represents a deadlock condition. There is a deadlock if old and new values are equal, thus a signal can be represented using a single variable, instead of two:

$$\left\{ \begin{array}{l} M = A \bmod B \\ A = M \end{array} \right. \quad \left\{ \begin{array}{l} M = A + B \\ A = M \end{array} \right.$$

Basically, the equation solving the first system is:

$$A = A \bmod B$$

The equation is verified for each value of A , such that $A < B$. The second system (loop) is represented by the following equation:

$$A = A + B$$

The equation does not have any solution, of course if $B \neq 0$.

The obtained results “formally” state that the first model enters a deadlock condition as soon as $A < B$, while the second model is never deadlocked.

The method has been explained using simple examples to let readers concentrate on the proposed technique rather than on managing more intricated graphs and bigger equation systems, that, due to space consideration, are out of the scope of this paper.

3 An analysis of VHDL Sequential Aspects

This section introduces a set of static analyses carried out on the VHDL code, focusing on the sequential aspects of the language. The aim of these analyses is to provide the user with information concerning the “future” testability of the circuit resulting from the synthesis of the considered code. When dealing with hardware testing issues, an important information relates to the presence of memory elements in the original circuit, which could be efficiently used for Scan purposes. The VHDL language allows the explicit declaration of memory elements in the structural style of description, allowing an immediate identification of these “test resources”. Yet, when considering a behavioral description, memory elements are often implicitly declared, deferring to a post-synthesis phase the use of such elements. One of the proposed analyses aims at an early detection of such implicit memory elements inferred from the synthesis process, by examining the VHDL code. This approach supplies information concerning the test phase after synthesis and before the circuit has been realized. This allows possible modifications in the device specification to improve the testability of the final device, right from its definition.

Still concerning testing issues, in terms of pattern application, the other analyses deal with the presence within the specified device of data paths such that data (test patterns and/or test results) can be transferred without being manipulated, thus constituting alternative accesses to module inputs/outputs for testing purposes. The advantage of dealing with data which is merely transferred rather than manipulated (added, incremented, ...) is trivial and relates to the minor complexity of test pattern generation and reduced problems caused by fault masking.

In the same scenario of test pattern/result propagation it is possible to introduce the last kind of analysis here proposed. It takes into account the explicit timing of the VHDL specification to determine the constraints that will need to be fulfilled to actually apply test patterns and access the generated results when dealing with modules part of a more complex system. In fact, in such a case it is important to evaluate the distance (in temporal terms) of the inputs/outputs of each module composing the complex system from the primary inputs and primary outputs which depends on the functionality and timings expressed by the VHDL code of the surrounding modules.

All these analyses provide the user with additional information concerning the circuit resulting from the synthesis process, so that possible improvements may be introduced from the testability point of view, which altogether relates to the quality of the specified device.

These analyses are all based on static analysis techniques, that, even if not as powerful as dynamic analysis, offer valuable results and are easier to apply.

All the techniques presented in the next sections are based on appropriate annotations on the nodes of a graph. As already done in Section 2, VHDL code is translated into a graph-like representation, called *flow graph*. A *flow graph* [19], basically a di-graph (directed graph), is composed of nodes, that represent program statements², and edges, that define execution flow. The following informal rules for constructing a *flow graph* can be listed:

- it must contain exactly one entry node which has no incoming arcs;
- a sequential statement is represented by a node with a single outgoing edge;
- an *if* statement is represented by a node with two outgoing edges, one for each execution thread;
- a *case* statement is represented by a node with as many outgoing edges as the alternatives listed;
- the end of a loop is represented by a node with two edges leaving it: one for exiting the loop, and the other one for going back to the first node of the loop.
- it must contain exactly one terminal node which has no arcs leaving it.

Notice that, for readers familiar with the LEDA tool [20], the representation proposed here is very similar to the graphs built by the tool.

The obtained graph is then decorated with the information needed for the particular analysis.

3.1 Implicit Memory Elements

This section sketches a way to find out implicit memory elements based on pattern searching in regular expressions. The starting point is the *flow graph* of a VHDL process, as defined in Section 3. For each variable, we define a regular expression summing up the significant actions done on the variable itself. Notice that, in this case, loops in *flow graphs* are not a problem. Remembering regular expression theory, they are simply translated by means of Kleene stars (*).

According to [18], given a VHDL process, a variable requires an implicit memory element each time:

²A “statement” can be both a single actual statement and a whole process invocation, considered as a macro-statement.

- it is read before being assigned.
- it is assigned before a `wait` statement.
- it is assigned in a clocked process.
- it is not assigned in all conditional branches.

This means that, as to this analysis, we need to take into account read, write (assignment), and wait operations. These actions correspond to a r , an a , and a w , respectively, within regular expressions. Thus ar means that the variable has been assigned a value and the value has been subsequently read. $w(r|a)$ states that, after the `wait` statement the variable is either read or assigned.

It should be clear now, that looking for implicit memory elements corresponds to looking for strings within regular expressions. The first two conditions above are easily translated by strings ra and aw . The third case means looking for a , but it applies to clocked processes only. Finally, the last rule states that, looking at possible alternatives, if one is an assignment, then all the alternatives must be assignments. Hence, the alternatives can be actually reduced to a single possibility. When the condition does not hold, an implicit memory element is needed.

Consider, as an example, the VHDL code of Figure 7 taken from [21]. It models a counter by defining a process with a clock signal and a reset, synchronized by means of a sensitivity list.

Clearly, the internal variable `count_int` included in a clocked process will be synthesized by means of a memory element, and the same holds true for output signals `zero` and `error` since, besides being part of a clocked process, their value is not updated for all possible conditional branches. These same results can be obtained by means of the proposed analysis of the regular expressions derived for the implied signals/variables. The regular expression for the variable `count_int` is $a|r(a|r|ra) = a|ra|rr|rra = a|ra|r$, where both a and r refer to A_C and R_C in the flow graph of Figure 8 to distinguish between different variables. The resulting regular expression straightfully matches the first condition. Signal `init_value`, which does not appear in the graph of Figure 8, since it constitutes a “parallel” graph with respect to it, does not match any of the four clauses, having regular expression a . Hence no memory element will be used.

Consider as another example the piece of VHDL code describing a demultiplexer (Figure 9). In this case the detection of inferred memory elements is not so trivial.

```

...
ARCHITECTURE behave OF count IS
BEGIN
  count_proc: PROCESS(clk, ld)
  VARIABLE count_int: t_value;
  BEGIN
    IF (ld = '1') THEN
      count_int := val;
      zero <= F0; -- forces 0 -
    ELSIF (clk = '1' AND clk'EVENT) THEN
      IF (dec = '1') THEN
        IF (count_int = 1) THEN
          count_int := 0;
          zero <= F1;
        ELSIF (count_int = 0) THEN
          error <= F1;
        ELSE
          count_int := count_int - 1;
        END IF;
      END IF;
    END IF;
  END PROCESS count_proc;
  init_value <= val;
END behave;

```

Figure 7: Counter: VHDL code (modified for synthesis with respect to the original)

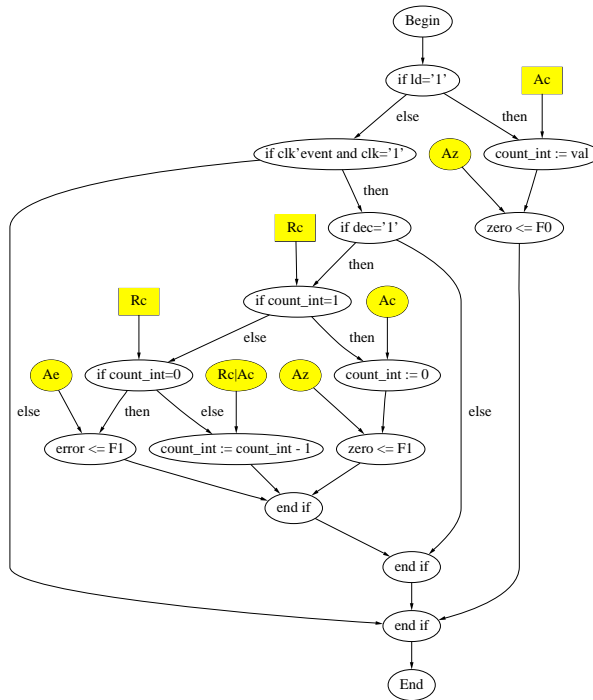


Figure 8: Counter: Flow Graph

```

ENTITY demux IS
  PORT(data_in: IN bit;
        sel: IN bit_vector(0 to 1);
        data_out: OUT bit_vector(0 to 3));
END demux;
ARCHITECTURE rtl OF demux IS
BEGIN
  PROCESS(data_in, sel)
    VARIABLE index: INTEGER;
  BEGIN
    index := TO_INTEGER('0' & sel, 0)
    data_out(index) <= data_in;
  END PROCESS;
END rtl;

```

Figure 9: Demultiplexer: VHDL code

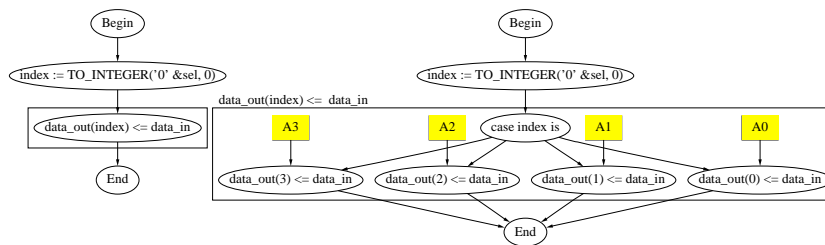


Figure 10: Demultiplexer: Flow Graph

The statement `data_out(index) <= data_in;` implies that one only of the outputs of `data_out` is assigned at each process execution. Such a situation corresponds to the fourth condition allowing the identification of implicit memory elements. This implies that four latches will be introduced during synthesis, one for each bit of the `data_out` vector. This conclusion may suggest the user the explicit definition of the memory elements, by introducing a register and logic for selecting the output on which to put the input value.

When dealing with vectors of data is thus necessary either to “explode” the statements, as depicted in the right flow graph, or to independently characterize each element of the vector with respect to the regular expressions.

Due to the nature of the analysis, it must be applied on single processes only. The result of the analysis identifies the presence of memory elements. The information can be used either to explicitly declare the register by rewriting the component specification in another style (e.g., structural description with instantiation of the register component and demultiplexer logic) or as an additional knowledge for test generation and management.

3.2 Data transfer path/elements

There are components, such as registers, multiplexers, demultiplexers, and decoders, whose functionality consists of data transferring without manipulating them. In particular, given an input signal, its value is *delivered* on the outputs, following some criteria. If dealing with registers, a delay is introduced and the input value is set on the outputs in a successive clock cycle with respect to when the input is read. In case multiplexers, demultiplexers, etc. are considered, either one of several different inputs is selected to be transferred to the output unmodified (multiplexer) or the input is transferred to one of the outputs while the others are set to a pre-defined value (demultiplexer; the one reported in Figure 9 stores the old value instead of using a pre-defined one).

For these variables/signals whose value is transferred from the inputs to the outputs without any manipulation the term “transferred” variable has been adopted, and it can be defined as a variable that is only read within a process: it does not appear in arithmetic or logic computations.

If a variable is transferred by a process, it can be used to propagate its test patterns through the process itself. Being able to statically identify which variables can be propagated through which processes provides more control on test patterns and thus is useful in testing hardware circuits.

Once more, let us consider the *flow graph*. Each node is associated with a set V of variables. All those variables for which the given definition of “transferred variables” does not hold anymore due to the statement in the node. Moreover, we define another set TV (Transferred Variables): it will contain the variables still transferred after each step.

At the beginning, TV contains all the variables of the process. After that, visiting each *flow graph* node, variables in V are subtracted from TV . At the end, i.e., after having visited all the nodes just once, TV contains the variables that are actually transferred.

In this case, we can extend the *flow graph* to cope with a set of related processes. Recall the counter process of Figure 7, input `val` is manipulated when considering its use in `count_int`, whereas the same `val` is not manipulated through `init_value`.

A node corresponds to a whole process. Its set V is simply the difference between all the variables and the set TV of the invoked process.

Let us consider different interacting processes, represented by their synthesized module in Figure 11. It is significant to be able to characterized transferred variables. For example, by analyzing the process describing the `Multiplexer`, variables `A`

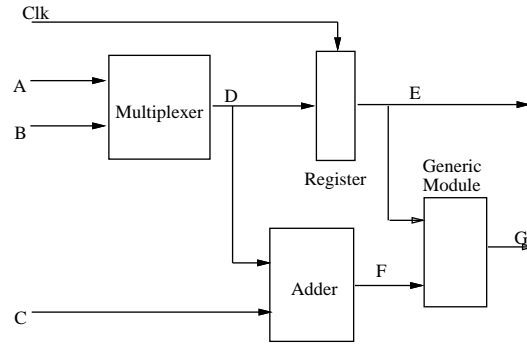


Figure 11: Processes (modules) interconnection and data transfer analysis.

and B are transferred to D. The same happens when analyzing the process defining the Register: variable D is transferred to E. As a consequence, variables A and B are transferred to E. On the other hand, variables A, B and C are not transferred to F. This analysis helps the user in managing test pattern generation when the Generic Module test set is considered, knowing that it can easily control values on E by acting directly on A and B, whereas determining the value of F is a more complex task.

3.3 Timing

When considering the possibility of applying the desired input patterns to a module embedded in a complex design, it is necessary not only to verify whether there is a direct path to the inputs under exam, but also what timing constraints are imposed by the upstream modules. Consider, as an example, the circuit depicted in Figure 12. To be able to apply any desired pattern to inputs E and F from the primary inputs (A and B) it is necessary to take into account that any value provided to P1 will be set to E after a delay of T1, whereas values on P2 will be available on input F after delay T2. This information may ease the test generation phase and may uncover possible problems of correlation in controllability and observability and synchrony during circuit testing [22]. In fact, it is statically possible to decide whether all the combinations of the possible values are observable on process inputs. Different delays could actually forbid some combinations.

An analysis that can be carried out on the *flow graph* of a VHDL process concerns the delays associated with variables, i.e., which is the actual effect of AFTER statements on variables propagation. Parametric values can be associated with arithmetic and logic operations to evaluate the delays through the flow graph. Furthermore, labels may be also used for clocked assignments which will require a time equal to a clock cycle to

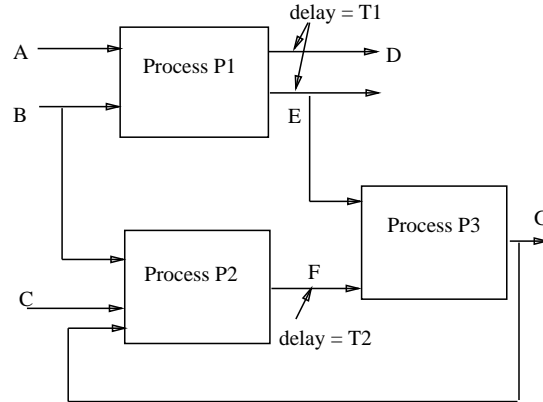


Figure 12: Timing analysis: module interconnection and timing constraints

provide the correct final value of assignments.

Before proposing an informal algorithm to evaluate the actual delays, we have to give the definition of *path*. A *path* is a directed path from the entry node to the terminal node of the *flow graph*. The presence of loops within a *flow graph* leads to have, at least from a theoretical point of view, an infinite set of *paths*. Fortunately, the problem can be overcome by exploiting both the peculiarities of VHDL and the specific needs of the current analysis. To have a specification that could be synthesized, loops must be upperbounded. The aforementioned assumption would definitively solve the problem. Moreover, to have a synthesizable specification, delays cannot be put within loops. This means that, as far as current analysis is concerned, loops can be discarded and the graph becomes a directed acyclic graph (DAG).

Graph nodes, in which variables are assigned, are annotated with a pair $\langle N, LD \rangle$. N is the variable name and LD is the local delay. LD is 0 if the variable is not delayed.

For each *path*, delays associated with each variable are summed, defining new pairs $\langle N, PD \rangle$. N is still the variable name and PD is the delay on the *path*. At the end, for each variable, the minimum and maximum values of PD define its delay domain. If the two values are 0, the variable is actually not delayed.

The analysis of all leaf processes is the precondition to be able to reason on a set of related processes. In this case, the *flow graph* contains nodes that are whole processes. These nodes are not associated with a single pair, but with a set of pairs: the results of the analysis on single processes. The set comprises a tuple $\langle N, LDmin, LDmax \rangle$ for each significant variable. Again, N is the variable name, $LDmin$ and $LDmax$ are the minimum and maximum delay evaluated for the variable. After that, the analysis goes on in almost the same way. To enrich the analysis and take into account the levels

```

...
alu: PROCESS(a, b, select)
BEGIN
  CASE select IS
    WHEN 0 => out <= a;           -- no delay
    WHEN 1 => out <= a + b;       -- delay delta_sum
    WHEN 2 => out <= a and b;     -- delay delta_gate
    WHEN 3 => out <= !a;         -- delay delta_gate
    WHEN 4 => out <= a or b;     -- delay delta_gate
    WHEN OTHERS => out <= a;    -- no delay
  END CASE;
END PROCESS alu;

```

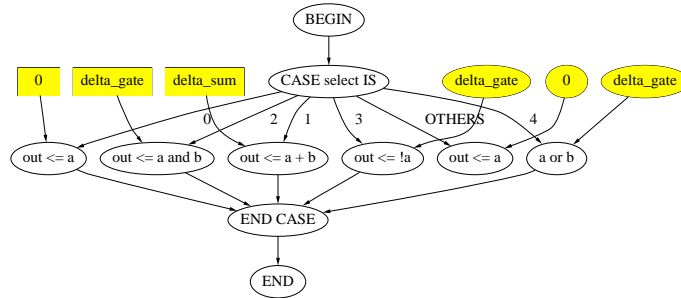


Figure 13: ALU: VHDL code and Flow graph

of logic to be traversed by the data, parametric information is also associated with arithmetic and logic operations. Consider the ALU specified in Figure 13. The input value of a (considered as a multi-bit signal) is immediately propagated to the output if $select$ is equal to 0 greater than 4. Two other delays are identified: one associated with the adder (multi-level logic) and the other for the one level of logic for AND/OR or complementing operations.

Figure 14 presents the VHDL code of a multiplexer, taken from [21]; the corresponding flow graph is shown in Figure 15. In this example, it can be noticed at a first glance that output Q is always deferred of 10 nsecs, whereas output $MUXSEL$ is never delayed. The proposed analysis provides the same information. If we consider annotated nodes only, i.e., the ones in which variable Q is assigned, we identify four paths. In all cases the corresponding pair is $\langle Q, 10 \rangle$. Hence Q is always deferred of 10 nsecs. Do note that statement $MUXSEL \leq muxval$ is concurrent with respect to the CASE statement.

3.4 A Complete Example

As a final example consider the VHDL code specification of a part of the Vending Machine [21] benchmark, opportunely modified to make it synthesizable, presented in

```

mux_proc: PROCESS(I0, I1, I2, I3, A, B)
  VARIABLE muxval : INTEGER range 0 to 4 := 0;
BEGIN
  muxval := 0;
  IF (A = '1') THEN
    muxval := muxval + 1;
  END IF;
  IF (B = '1') THEN
    muxval := muxval + 2;
  END IF;
  MUXSEL <= muxval;
  CASE muxval IS
    WHEN 0 => Q <= I0 after 10 ns;
    WHEN 1 => Q <= I1 after 10 ns;
    WHEN 2 => Q <= I2 after 10 ns;
    WHEN 3 => Q <= I3 after 10 ns;
    WHEN OTHERS => Q <= 0 after 10 ns;
  END CASE;
END PROCESS mux_proc;

```

Figure 14: Timing analysis: VHDL code

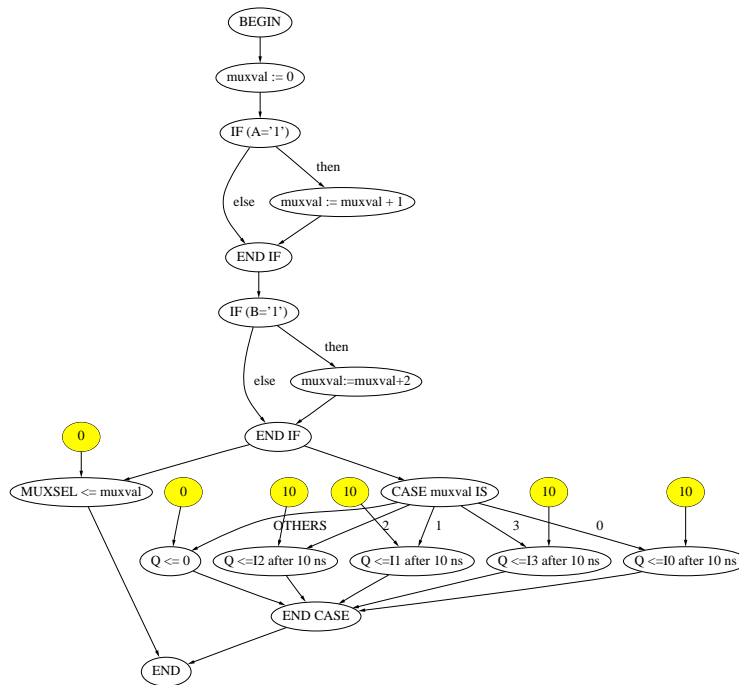


Figure 15: Timing analysis: flow graph.

Figure 16.

It is possible to apply all the three kinds of proposed analyses to the piece of code in Figure 16. The process `change_proc` is clocked, so all the variables and signals that appear as target for assignment statements will be surely implemented by means of memory elements. The `display` signal, instead, is characterized by regular expression `a` and is not included in the clocked process, so no memory element will be used. Furthermore, signals `total`, `coin_reject` and `error` have regular expressions such that not for all conditional branches an assignment is made. The analysis allows the user to highlight this aspect of the written specification for eventual modification if desired.

By analyzing the presence of “data transfer” variables, all variables set on the outputs are manipulated within the process except `price`, that is directly set on the `display` output.

Finally, as far as the timing characterization is concerned, it is possible to state that `total` is the only delayed variable due to arithmetic manipulation affecting `int_total`. Variables `coin_reject` and `error` are not delayed but characterized by a clocked temporization, as well as `total`, whereas `display` is neither delayed nor clocked. Hence, the downstream modules, taking as inputs the signals generated by the `change_proc` will have to consider such a timing constraints affecting the possible combinations of values.

4 Conclusions and Future Work

The short review presented so far does not aim at listing all the possible applications of software validation methodologies to the analysis of VHDL behavioral descriptions. It is only a first overview that both helped us in thinking about interrelations between the two domains, and could provide a first basis for further proposals.

The work did not pay too much attention to dynamic analysis methods [23] for two main reasons. There already exist tools specific to VHDL that supply facilities for coverage analysis [24, 9, 25]. Moreover, testing of VHDL code usually copes with fault models [26]. These models are not easy to deal with for not expert users, and heavily impact on obtained results.

The techniques in this paper gave encouraging results as to the case studies used to validate them.

As a side effect, the work revealed also the two different finalities of testing a software program and a VHDL behavioral specification. In the former case, testing is used

```

...
ARCHITECTURE behave OF coin_handler IS
BEGIN
  change_proc: PROCESS(clk, reset)
  VARIABLE local_change : INTEGER := 0;
  VARIABLE int_total : INTEGER := 0;
  BEGIN
    IF (reset = '1') THEN
      int_total := 0;
      total <= 0;
    ELSIF (clk = '1' AND clk'event) THEN
      IF (sell_en = '1') THEN
        IF (coin_stb = '1') THEN
          IF (int_total >= max_price) THEN
            coin_reject <= F1;
          ELSE
            coin_reject <= F0;
            int_total := int_total + coin_in; -- delta_add --
            total <= int_total;
          END IF;
        ELSIF (change_stb = '1') THEN
          local_change := change_in;
          IF (int_total >= local_change) THEN
            int_total := int_total - local_change; -- delta_sub --
            total <= int_total;
            IF (int_total >= max_price) THEN
              coin_reject <= F1;
            ELSE
              coin_reject <= F0;
            END IF;
          ELSE
            error <= F1;
          END IF;
        ELSIF (item_out_stb = '1') THEN
          IF (int_total >= price) THEN
            int_total := int_total - price; -- delta_sub --
            total <= int_total;
            IF (int_total >= max_price) THEN
              coin_reject <= F1;
            ELSE
              coin_reject <= F0;
            END IF;
          ELSE
            error <= F1;
          END IF;
        END IF;
      END IF;
    END PROCESS change_proc;

    display <= price;
  END behave;

```

Figure 16: Coin_Handler: VHDL code and Flow graph

to validate produced code, i.e., to try to correct possible errors in the implementation. In the latter case, testing is not employed to uncover problems, lacks or inconsistencies, but to generate test patterns which will be used during circuit validation [27, 28]. Basically testing means producing results for further activities, instead of validating what already done.

Finally, we want to suggest other possible contributions of software techniques to the analysis of VHDL code:

- Pattern matching techniques could be used on VHDL code to identify known hardware topologies and to provide guidelines for designers [29]. Considering again *flow graphs* as suitable representations of VHDL, the reach theory of graph-grammars [30] could help us in carrying out this task.
- One of the branches of software testing is mutation analysis [31]. A program is verified by means of all its mutations. A mutation is obtained from the original program by injecting a syntax mutation that leads to a change in program semantics. It sounds very similar to faults models introduced in VHDL analysis: it could be tried to define the mutations of a VHDL specification and to study the relations with fault models.
- Software testing uses coverage criteria [32] to determine how much code is actually executed given a test case. VHDL testing generates test patterns from behavioral specification. Putting together the two things, test patterns could be used as test cases for coverage criteria. It could be studied if there exist some relations between the results gained from coverage criteria and the completeness of circuit validation.

References

- [1] P.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988
- [2] J.B. Goodenough and S.L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.
- [3] M. Pezzè, R.N. Taylor, and M. Young. Graph Models for Reachability Analysis of Concurrent Programs. *ACM Transactions on Software Engineering and Methodology*, 4(2):171–213, 1995.
- [4] A. Balboni, M. Mastretti, and M. Stefanoni. Static Analysis of VHDL Model Evaluation. In *Proceedings of Euro-VHDL*, pages 586–591, 1994.

- [5] S. Carlson and E. Girczyc. Increasing Design Quality and Engineering Productivity through Design Reuse. In *Proceedings of 30th Design Automation Conference*, 1993.
- [6] O. Levia. Writing High Performance VHDL Models. In *Proceedings of Euro-VHDL*, 1991.
- [7] X. Gu, K. Kuchcinski, and Z. Peng. Testability Analysis and Improvement from VHDL Behavioral Specifications. In *Proceedings of EURO-VHDL '94*, pages 644–649, 1994.
- [8] Synopsys. *Synopsys User's Manual*, 1994.
- [9] Veda. *VHDL Cover User's Manual*.
- [10] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. Symbolic Execution of Concurrent Systems using Petri Nets. *Computer Languages*, 14(4):263–281, 1989.
- [11] A.S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall International Editions, 1992.
- [12] Mentor Graphics. *Autologic VHDL Reference Manual*, 1993. version 8.2.
- [13] E.W. Dijkstra. *Co-operating Sequential Processes*. Academic Press, 1965.
- [14] J.C. Corbett. Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.
- [15] G.M. Karam and R.J. Buhr. Starvation and Critical Race Analyzers For Ada. *IEEE Transactions on Software Engineering*, 16(8):829–843, 1990.
- [16] A. Valmari. A Stubborn Attack on State Explosion. *Computer Aided Verification '90*, 3:25–41, 1991.
- [17] R.N. Taylor, D.L. Levine, and C.D. Kelly. Structural Testing of Concurrent Programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, 1992.
- [18] J. Bergé, A. Fonkoua, S. Maginot, and J. Rouillard. *VHDL Designer's Reference*. Kluwer Academic Publishers, 1992.
- [19] M.S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, 1977.
- [20] LEDA S.A. *LEDA VHDL System - User's Manual*, 1993. version 3.2.0.
- [21] D.L. Perry. *VHDL*. McGraw-Hill.
- [22] M. Bombana, G. Buonanno, P. Cavalloro, F. Ferrandi, D. Sciuto, and G. Zaza. ALADIN: A Multi-Level Testability Analyzer for VLSI System Design. *IEEE Transactions on VLSI Systems*, 2(2):157–171, 1994.
- [23] W.E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill, 1987.
- [24] Synopsys. *VSS User's Manual*.
- [25] Vantage. *CodePerfect User's Manual*.

- [26] S. Ghosh and T.J. Chakraborty. On Behavior Fault Modeling for Digital Designs. *Journal of Electronic Testing: Theory and Applications*, 2(2):135–151, 1991.
- [27] J.F. Santucci, A.L. Courbis, and N. Giambiasi. Behavioral Testing of Digital Circuits. *Journal of Microelectronic Systems Integration*, 1(1):55–77, 1993.
- [28] S.R. Rao, B. Pan, and J.R. Armstrong. Hierarchical Test Generation for VHDL Behavioral Models. In *Proceedings of Euro-DAC*, pages 175–182, 1993.
- [29] C. Bolchini. A VHDL Description for a Design for Testability Approach. Technical report, Politecnico di Milano, 1995.
- [30] A.C. Shaw. Parsing of Graph-Representable Pictures. *Journal of the ACM*, 17(3):453–581, 1970.
- [31] R. DeMillo, R.J. Lipton, and F.G. Sayward. Program Mutation: A New Approach to Program Testing. Technical report, Infotech International, 1979.
- [32] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990. Second Edition.